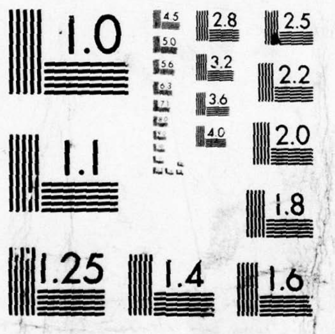
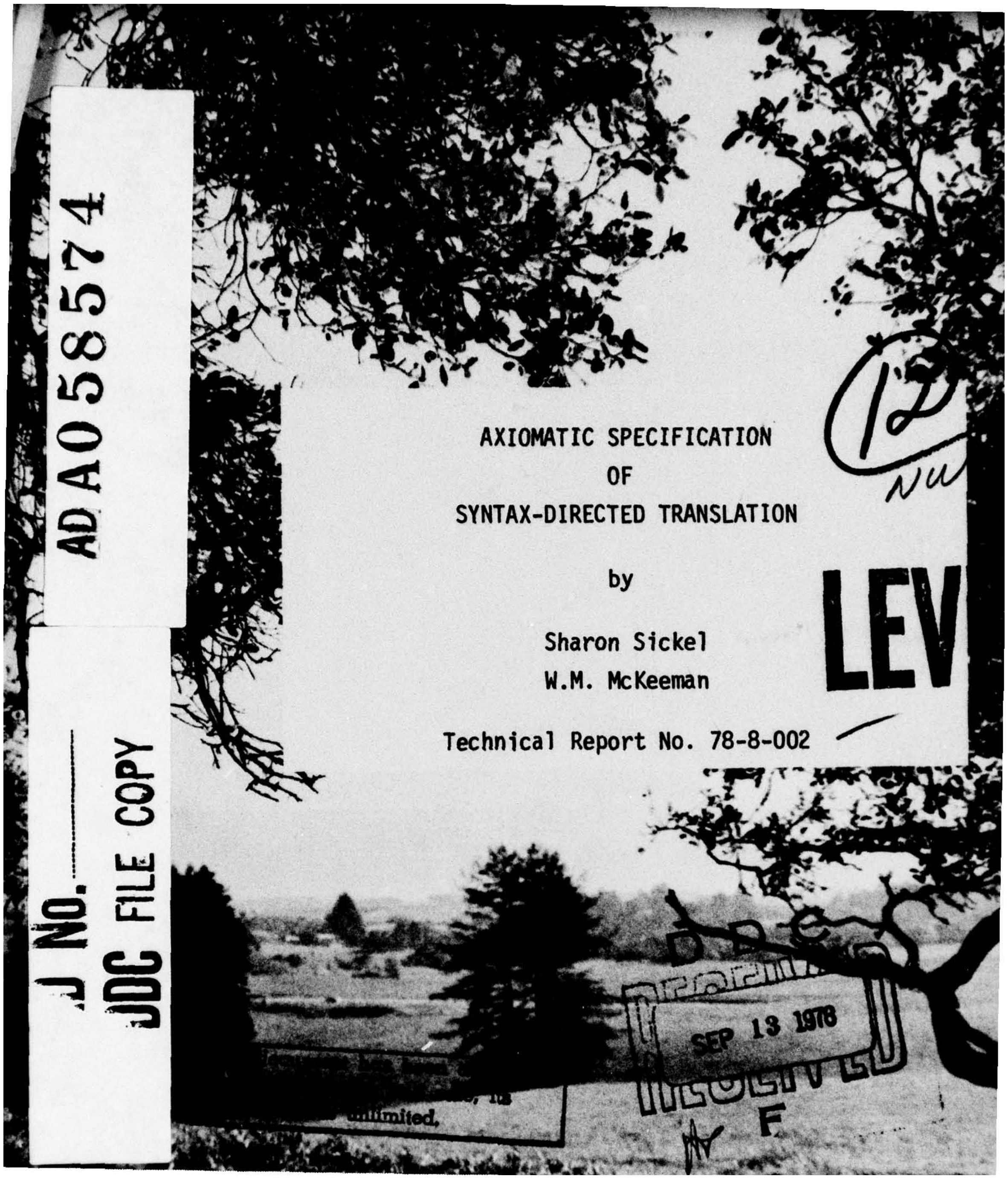
AD-A058 574 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
AXIOMATIC SPECIFICATION OF SYNTAX-DIRECTED TRANSLATION.(U)
AUG 78 S SICKEL, W M MCKEEMAN N00014-76-C-0682
UNCLASSIFIED TR-78-8-002 NL

| OF |
AD
A058574

END
DATE
FILMED
11-78
DDC


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6



12
NU

AXIOMATIC SPECIFICATION OF SYNTAX-DIRECTED TRANSLATION

by

Sharon Sickel
W.M. McKeeman

LEV

Technical Report No. 78-8-002

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER
-

4. TITLE (and Subtitle)
6 AXIOMATIC SPECIFICATION OF SYNTAX-DIRECTED TRANSLATION.

5. TYPE OF REPORT & PERIOD COVERED
9 Technical rept.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
10 Sharon Sickel and W.M. McKeeman

8. CONTRACT OR GRANT NUMBER(s)
15 N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Information Sciences
University of California
Santa Cruz, California 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE
August 1, 1978

13. NUMBER OF PAGES
16

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

11 1 Aug 78 12 30p.

18. SUPPLEMENTARY NOTES

14 TR-78-8-002

19. KEY WORDS (Continue on reverse side if necessary and identify by block number) and Phrases:

Syntax-directed translation, context-free grammars, logic programming, program specification.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Predicate logic is applied to the specification of syntax-directed translation. Context-free grammars are shown to be representable by logic programs, and a translation from grammars to logic programs is presented as a logic program. Coupled grammars are introduced, shown to be interpretable as translators, shown to be representable as logic programs, and translation from coupled grammars to logic programs is presented via logic programs. Mixed grammars, a concisely representable special case of coupled grammars, are given in terms of coupled grammars.

78 09 08 023

410 350

AXIOMATIC SPECIFICATION

OF

SYNTAX-DIRECTED TRANSLATION

by

Sharon Sickel

W.M. McKeeman

Information Sciences and Crown College

University of California

Santa Cruz, Ca. 95064

This work was supported by Office of Naval Research Contract # 76-C-0682

ABSTRACT

Predicate logic is applied to the specification of syntax-directed translation. Context-free grammars are shown to be representable by logic programs, and a translation from grammars to logic programs is presented as a logic program. Coupled grammars are introduced, shown to be interpretable as translators, shown to be representable as logic programs, and translation from coupled grammars to logic programs is presented via logic programs. Mixed grammars, a concisely representable special case of coupled grammars, are given in terms of coupled grammars.

1. Introduction

We define grammars to be coupled iff they are context-free and there is a 1-1 correspondence between their productions. Two different interpretations are given for them. The first is a generalization of the syntax-directed transductions of Lewis and Stearns [2]. Their productions take the form

$$A \to \alpha, \beta$$

where the parts of the form

$$A \to \alpha$$

constitute the underlying input grammar, and the string β specifies the output. The nonterminals of each β must be a permutation of those in the corresponding α. As $A \to \alpha$ is applied during the parse, the value of A is given by β where each nonterminal is replaced by its own value. Our paired productions take the form

$$A \to \alpha \quad \text{and} \quad A \to \beta$$

but we relax the requirement of a 1-1 correspondence between nonterminals in α and β. A missing nonterminal in β is ignored, its value being discarded. An extra nontermin in β is kept literally, standing in the output for any string derivable from it by th productions in the second grammar.

The second interpretation of coupled grammars links input to output via the pars sequence. Aho and Ullman [1, Vol. II] present a similar scheme based on the production form

$$A \rightarrow \alpha, \beta$$

where again the nonterminals in α and β must correspond. Our productions take the form

$$A \rightarrow \alpha \quad \text{and} \quad B \rightarrow \beta$$

where there are no restrictions on the form or correspondence of symbols within the productions.

The logic program corresponding to a grammar provides a static statement of the meaning of the grammar; that is, an axiomatic specification of it. Each predicate in the logic program specifies a relation among its arguments.

Mixed grammars are a concise notation for the simple syntax-directed translation schemata presented by Aho and Ullman [1, Vol. I]. There is a nonterminal vocabulary V_N and two terminal vocabularies, V_I for the input and V_O for output. Mixed grammars are a directly executable notation on a simple pushdown stack machine. They have a concise representation in logic. They are, furthermore, self-describing, and self-translating.

2. Logic Representations of Grammars

A context-free grammar G is a four-tuple $[V_N, V_T, P, S]$ as usual. P is a set of productions of the form

$$A = \alpha$$

where we have used the symbol = instead of $\rightarrow$ to avoid confusion with $\rightarrow$ for implication.

$$A \in V_N \quad \text{and} \quad \alpha \in (V_N \cup V_T)^*.$$

The empty string is denoted by ε. We assume a function, start(G) = S. We further follow the conventions that nonterminals are denoted by capital letters and terminals by any other symbols except Greek letters which we sometimes use to denote strings. Because of these conventions and the existence of the function, start, we can completely specify a grammar as a sequence of productions, and we frequently do.

A logic program [4] is a set of WFs in the form

$$L \leftarrow R$$

and one WF, the call, of the form

$$\leftarrow R$$

where L is a predicate and R is a conjunction of predicates. All variables are implicitly universally quantified.

Table 2.1 is an example of a context-free grammar and a corresponding logic program. Terminal symbols in the grammar are delimited by double quotes, and the parameters to the predicates are strings of symbols represented by the juxtaposition of variable names and constants delimited by double quotes. We may drop the quotes when the meaning is obvious.

G = RG	G(x1 x2) ← R(x1) ∧ G(x2)
G = R	G(x1) ← R(x1)
R = L"="F	R(x1 "=" x2) ← L(x1) ∧ F(x2)
F = PF	F(x1 x2) ← P(x1) ∧ F(x2)
F = P	F(x1) ← P(x1)
P = """U"""	P(""" x1 """) ← U(x1)
P = L	P(x1) ← L(x1)
P = U	P(x1) ← U(x1)
L = "G"	L("G") ←
L = "R"	L("R") ←
L = "F"	L("F") ←
L = "P"	L("P") ←
L = "L"	L("L") ←
L = "U"	L("U") ←
U = L	U(x1) ← L(x1)
U = "="	U("=") ←
U = """	U(""") ←

Table 2.1: Grammar GG and its logic program representation LPGG.

The interpretation of the grammar GG is obvious [3]. It is in fact self-describing. The correspondence between each production of grammar and Horn clause is straightforward. For example,

$$G(x1\ x2) \leftarrow R(x1) \wedge G(x2)$$

states that if R is true on string x1 and G is true on string x2, then G is necessarily true on the concatenation of x1 and x2, which is also the meaning of

$$G = RG$$

in the grammar.

$$L(\text{"G"}) \leftarrow$$

states that L is always true for the letter G, and so on.§

3. Translation from Context-free Grammars to Their Logic Interpretations

Predicate TGL formally defines the relationship between context-free grammars and their interpreting logic programs, and also the translation process between the two, i.e., TGL(G,LP) is true iff LP is the logic program representation of grammar G. In the example of Table 2.1, TGL(GG,LPGG).

$$\begin{aligned}
&TGL(\varepsilon,\varepsilon) \leftarrow \\
&TGL((A = \beta)\cdot G,\ New\_rule\cdot LP) \leftarrow \\
&\qquad TGL(G,LP) \\
&\quad \wedge\ TPL(\beta,1,Iparam,Conj) \\
&\quad \wedge\ New\_Rule = (A\ \text{"("}\ Iparam\ \text{")} \leftarrow \text{"}\ Conj)^{\ddagger}
\end{aligned}$$

§ In examples we occasionally optimize logical formulas to remove the value true where it is formally present but contributes no meaning. In particular $A \wedge true \Leftrightarrow A$ and $(A \leftarrow true) \Leftrightarrow (A \leftarrow)$.

‡ The symbol "·" appears frequently in parameters of logic programs. u·v denotes a parameter that is subdivided into components u and v. For strings we denote such a decomposition simply by juxtaposition of symbols, e.g. uv. The use of "·" denotes decomposition of more complicated structures, such as grammars. The first use here, $(A = \beta)\cdot G$, denotes a grammar that is subdivided into its first production $A = \beta$ and the remaining productions, grammar G.

Given β, the r.h.s. of a production, TPL(β,1,Iparam,Conj) yields both a parameter list, Iparam, and a conjunction of predicates, Conj. More specifically TPL(β,n,Iparam,Conj) is true iff $\beta = \beta_n B_n \beta_{n+1} \ldots B_k \beta_{k+1} \in (V_N \cup V_T)^*$ where each $\beta_i \in V_T^*$ and $B_i \in V_N$ and Iparam = $\beta_n x_n \beta_{n+1} \ldots x_k \beta_{k+1}$, and Conj = $B_n(x_n) \wedge B_{n+1}(x_{n+1}) \wedge \ldots B_k(x_k)$.

```
TPL(ε,n,ε,true) ←
TPL(s β,n,s Iparam,Conj) ← T(s)
        ∧ TPL(β,n,Iparam,Conj)
TPL(S β,n,X(n) Iparam,S "(" X(n) ") ∧ " Conj) ← N(S)
        ∧ TPL(β,n+1,Iparam,Conj)
```

T(s) is true iff s is a terminal symbol.

N(S) is true iff S is a nonterminal symbol.

X(n) is the letter x subscripted with the value of n, and similarly for Y(n) which will be used later.

4. Functional Interpretation of Coupled Grammars

Coupled grammars have an input grammar and an output grammar. As each production of the input grammar is used, an output is associated with it as specified by the corresponding production of the output grammar. This relation can be specified by a logic program as shown in Table 4.1 for coupled grammars B and U translating binary to unary notation.

grammar B	grammar U	logic program LPBU
A = A 1	A = AA 1	$A(x_1\ 1, y_1 y_1\ 1) \leftarrow A(x_1, y_1)$
A = A 0	A = AA	$A(x_1\ 0, y_1 y_1) \leftarrow A(x_1, y_1)$
A = 1	A = 1	$A(1,1) \leftarrow$
A = 0	A = ε	$A(0,\varepsilon) \leftarrow$

Figure 4.1: Coupled grammars B and U and their logic program representation LPBU.

It is obvious that the input grammar describes all binary strings, and the output grammar (ambiguously) describes all unary strings. The translation doubles the value of the output string for each shift in positional notation and adds a further "1" if the number is odd.

The logic program predicates have two parameters, one for input and one for output. For example, the interpretation of the clause

$$A(x_1\ 1, y_1 y_1\ 1) \leftarrow A(x_1, y_1)$$

says that if x_1 is an acceptable string producing output y_1, then x_1 1 is an acceptable string producing $y_1 y_1$ 1 as output.

5. Translation of Coupled Grammars to Their Functional Interpretation

Given coupled grammars G_1 and G_2, one can construct a program that will perform a functional interpretation on a string, using G_1 as the input grammar and G_2 as the output grammar. $FI(G_1,G_2,LP)$ is true iff LP, a logic program, performs that interpretation. For example, from Figure 4.1, FI(B,U,LPBU).

$$\begin{aligned}
&FI(\varepsilon,\varepsilon,\varepsilon) \leftarrow \\
&FI((A = \beta)\cdot G_1,(A = \gamma)\cdot G_2,z\cdot LP) \leftarrow \\
&\qquad FI(G_1,G_2,LP) \\
&\qquad \wedge\ FIP(A,\beta,\gamma,z)
\end{aligned}$$

$FIP(A,\beta,\gamma,C)$ means that the functional interpretation of coupled productions $A = \beta$ and $A = \gamma$ is the logic procedure C where A is a single nonterminal and β and γ are strings in V^*. In the program below, Iparam and Oparam are each strings of variables and terminals and represent the input parameter and output parameter, respectively. R is a conjunction of predicates.

```
FIP(A,β,γ,A "(" Iparam "," Oparam ") ← " Conj) ←
        IN(β,1,Iparam,Conj)
      ∧ OUT(Conj,γ,Oparam)
```

IN(β,n,Iparam,Conj) serves a similar function to TPL(β,n,Iparam,Conj) except that we are producing two parameters instead of one to the predicates of Conj, i.e. whenever $P(x_i)$ appears in Conj of TPL, then $P(x_i,y_i)$ appears in Conj of IN.

```
IN(ε,n,ε,true) ←
IN(s β,n,s Iparam,Conj) ← T(s)
      ∧ IN(β,n,Iparam,Conj)
IN(S β,n,X(n) Iparam,S "(" X(n) "," Y(n) ") ∧ " Conj ← N(S)
      ∧ IN(β,n+1,Iparam,Conj)
```

OUT(Conj,γ,Oparam): Oparam is the output parameter and if $\gamma = \gamma_1\gamma_2\ldots\gamma_n$ then Oparam denotes

$s_1 s_2 \ldots s_n$ where

$s_i = \gamma_i$ if $t(\gamma_i)$

$= y_j$ if $N(\gamma_i)$ and j is the least integer such that $\gamma_i(x_j,y_j)$ is a literal in Conj

$= s_i$ if $N(\gamma_i)$ and $\gamma_i(x_j,y_j)$ is not in Conj

```
OUT(Conj,ε,ε) ←
OUT(Conj,s γ,s Oparam) ← T(s)
      ∧ OUT(Conj,γ,Oparam)
OUT(Conj,S γ,y Oparam) ← N(S)
      ∧ FIND(S,Conj,y)
      ∧ OUT(Conj,γ,Oparam)
```

FIND(S,Conj,z): For nonterminal S, and conjunct Conj,

z = y for leftmost predicate in Conj that is of the form S(x,y)

= z if no such predicate exists in Conj

FIND(S,true,S) ←

FIND(S,S "(" x "," y ") ∧ " Conj,y) ←

FIND(S,R "(" x "," y ") ∧ " Conj,z) ← (S≠R)

∧ FIND(S,Conj,z)

For example, if A, B, C and D are nonterminals and # and ! are terminals, then the following are true:

$\text{FIP}(A,\#BCD,CB!,\ A(\#x_1x_2x_3,y_2y_1!) \leftarrow B(x_1,y_1) \wedge C(x_2,y_2) \wedge D(x_3,y_3))$

$\text{IN}(\#BCD,\ 1,\ \#x_1x_2x_3,\ B(x_1,y_1) \wedge C(x_2,y_2) \wedge D(x_3,y_3))$

$\text{OUT}((B(x_1,y_1) \wedge C(x_2,y_2) \wedge D(x_3,y_3)),\ CB!,\ y_2y_1!)$

6. Parse Sequence Interpretation of Coupled Grammars

Given a grammar G, with each string in L(G) is associated one (or more) parse sequences. A parse sequence is a sequence of integers corresponding to the production numbers as they are applied in a left-to-right parse.

Suppose we have two arbitrary coupled grammars, and each is used to parse a string in its language. The strings are defined to be equivalent if they have a parse sequence in common, as shown in Figure 6.1.

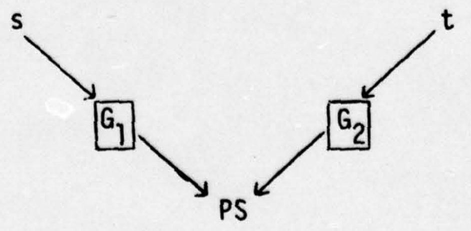


Figure 6.1: Parse sequence interpretation with the parse sequences not used intermediately.

Alternatively, take the same two coupled grammars, where the input grammar is used to generate a parse sequence and then that parse sequence is used with the output grammar to generate output as shown in Figure 6.2. We have then defined a means of translation.

$$\begin{tikzcd}
s \arrow[d] \\
\boxed{G_1} \arrow[d] \\
PS \arrow[d] \\
\boxed{G_2} \arrow[d] \\
t
\end{tikzcd}$$

Figure 6.2: Parse sequence interpretation with the parse sequence used as an *intermediate form.*

E = E + T	P = P P +
E = T	P = P
T = T * D	P = P P *
T = D	P = P
D = (E)	P = P
D = 0	P = 0
D = 1	P = 1
D = 2	P = 2
D = 3	P = 3
⋮	⋮
D = 9	P = 9
input grammar	output grammar

Table 6.1: Parse sequence coupled grammars.

The coupled grammars in Table 6.1 relate infix expressions and Polish expressions. The input grammar is unambiguous; every parse sequence from it is a parse sequence for the output grammar. Thus, there is a Polish form for every infix expression. Because of rules P = P in the output grammar, it is ambiguous. For every Polish string there are infinitely many parses which are also parses for the input grammar. Each defines a correct translation. There are also parses from the output grammar which are meaningless relative to the input grammar.

7. Translation of Coupled Grammars to Their Parse Sequence Interpretation

PSI(G_1,G_2,s,t) is true iff grammar G_1 parsing string s and grammar G_2 parsing string t give the same parse sequence. We give two different definitions for PSI. The first generates an explicit parse sequence from an input string and its relevant grammar, then uses that sequence to generate a corresponding string in the language of the other grammar.

```
PSI(G1,G2,s,t) ←
        PARSE(G1,s,PS)
      ∧ GENERATE(G2,PS,t)
```

PARSE(G_1,s,PS) is true iff PS is a parse sequence of string s in grammar G_1. The parse is accomplished by creating an associated grammar GS, then using logic program FI to translate G_1 and GS into a logic program that carries out a functional interpretation between the grammars, and, finally, to execute that logic program with string s as input to produce parse sequence PS.

```
PARSE(G1,s,PS) ←
        SEQ(G1,1,GS)
      ∧ FI(G1,GS,LP)
      ∧ EXEC(("←" Start(G1) "(" s "," PS ")" )
            ·LP)
```

SEQ(G_1,n,GS) is true iff G_1 and GS are context-free grammars, n is a positive integer, and by considering the productions of G_1 sequenced starting at n, GS consists of corresponding productions in which each r.h.s. is the non-terminals, in order, of its corresponding production in G_1, followed by the sequence number of that production.

Formally, SEQ is defined:

$SEQ(\varepsilon,n,\varepsilon) \leftarrow$

$SEQ((A = \alpha)\cdot G,n,(A = \beta n)\cdot GS) \leftarrow$

$\quad SEQ(G,n+1,GS)$

$\quad \wedge\ STRIP(\alpha,\beta)$

STRIP(α,β) is true iff α is a string in V*, and β is α with terminals stripped away.

$STRIP(\varepsilon,\varepsilon) \leftarrow$

$STRIP(s\ \alpha,\beta) \leftarrow T(s)$

$\quad \wedge\ STRIP(\alpha,\beta)$

$STRIP(S\ \alpha,S\ \beta) \leftarrow N(S)$

$\quad \wedge\ STRIP(\alpha,\beta)$

EXEC(P) is a meta-procedure that executes logic program P. For an example of PARSE, consider

G_1:	s:
E = E+T	a*a + a*a
E = T	
T = T*a	
T = a	

the calls and computed values are:

SEQ(G_1,1,GS) in which GS becomes:

E = ET1
E = T2
T = T3
T = 4

FI(G_1,GS,LP) in which LP becomes:

$E(x_1"+"x_2, y_1y_2\ 1) \leftarrow E(x_1,y_1) \wedge T(x_2,y_2)$
$E(x_1,y_1\ 2) \leftarrow T(x_1,y_1)$
$T(x_1"*"a, y_1y_2\ 3) \leftarrow T(x_1,y_1)$
$T(a,4) \leftarrow$

EXEC(←E("a*a+a*a",PS) · LP) in which PS becomes:

432431

Now this completes the first half of the definition of PSI. Given the parse sequence constructed in the above process we can use it to drive a right-most derivation in G_2, to create string t.

We now define the predicate GENERATE. We know intuitively that parsing and generation of strings are inverse operations. That would say that we could define GENERATE in terms of PARSE, thus:

$$\text{GENERATE}(G_2,\text{PS},t) \leftrightarrow \text{PARSE}(G_2,t,\text{PS})$$

The way we have used PARSE (and think of parsing) is that the string in the language is given and we generate the parse sequence as a side-effect of the recognition process. Suppose the parse sequence and the grammar are given. Can we use PARSE to create the input? A useful property of logic programs is that they describe truth about relationships, and while they can drive computations, the direction of the computation is usually arbitrary. Let's follow the computation to see if the string t can be appropriately computed.

Continuing the example above, let G_2 be:

$P = P\ P\ +$
$P = P$
$P = P\ P\ *$
$P = a$

Then PARSE(G_2,t,432431) calls SEQ(G_2,1,GS') which produces GS':

$P = P\ P\ 1$
$P = P\ 2$
$P = P\ P\ 3$
$P = 4$

Then FI(G_2,GS',LP') computes LP':

$P(x_1x_2\ +, y_1y_2\ 1) \leftarrow P(x_1,y_1) \wedge P(x_2,y_2)$
$P(x_1,y_1\ 2) \leftarrow P(x_1,y_1)$
$P(x_1x_2\ *, y_1y_2\ 3) \leftarrow P(x_1,y_1) \wedge P(x_2,y_2)$
$P(a,4) \leftarrow$

Executing LP' with call ← P(t,432431), we compute t = aa*aa*+, the desired answer.

Therefore, we could have defined PSI as:

```
PSI($G_1$,$G_2$,s,t) ←
        PARSE($G_1$,s,PS)
      ∧ PARSE($G_2$,t,PS)
```

And, we see that our original claim of the equivalence of the two definitions is reflected in their having a single formal specification.

8. Mixed Grammars

Suppose we have a set of nonterminals V_N and two disjoint sets of terminals

V_I and V_O, and a context-free grammar

$$[V_N, V_I \cup V_O, P, S] .$$

The productions in P are of the form

$$A = \alpha$$

where

$$\alpha \in (V_N \cup V_I \cup V_O)^*.$$

They are equivalent to the coupled grammars

$$[V_N, V_I, P_I, S] \quad \text{and} \quad [V_N, V_O, P_O, S]$$

where all elements of V_O are deleted from the productions of P to give P_I and vice-versa for P_O.

The grammars can be interpreted either as functionally coupled or parse sequence coupled. They are equivalent to the simple syntax-directed translation schemata of Aho and Ullman [1, Vol. I].

Notationally speaking, it is convenient for $V_I \cap V_O$ to be nonempty, thus we establish the convention of double quotes delimiting the members of V_I (as in earlier sections of this paper) and single quotes delimiting the members of V_O.

```
E = E "+" T '+'
E = T
T = T "*" "a" 'a' '*'
T = "a" 'a'
```

Table 8.1: A mixed grammar describing the translation from infix to Polish.

For example, the mixed grammar in Table 8.1 has the same effect as the coupled grammars in the preceding examples. It will also accept other strings but its behavio
is then of no interest.

The advantages of mixed grammars are that they are directly executable on a simple pushdown store machine, and that their notation makes implicit, unavoidable and natural the constraints for simple syntax-directed translation schemata. One can produce a logic program similar to PSI and FI to translate mixed grammars to logic.

9. Conclusions

We have established some relationships among context-free grammars, translation schemata, and logic. The interpretation of paired grammars has been extended in several ways. We have defined a set of translation programs that are actually sets of logic theorems. They are concise, and the correctness of each program can be established by proving each theorem individually. The process of parsing and the process of generation which are inverses are shown to have the same formal specification.

REFERENCES

1. Aho, A. E. and J. D. Ullman. The Theory of Parsing, Translation and Compiling, Vols. I & II, Prentice-Hall (1972,1973), especially Chapters 3, 6 & 9.

2. Lewis, P. M. II and R. E. Stearns. Syntax-Directed Transduction, J.ACM 15:3 (July 1968), pp. 464-488.

3. McKeeman, W. M. Concise Extensible Translator Notations, Technical Report No. 78-8-001, Information Sciences (August 1978).

4. Van Emden, M. H. and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language, J.ACM 23:4 (October 1976), pp. 733-742.

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy